DFL: Secure and Practical Fault Localization for Datacenter Networks

Xin Zhang, Fanfu Zhou, Xinyu Zhu, Haiyang Sun, Adrian Perrig, *Senior Member, IEEE*, Athanasios V. Vasilakos, *Senior Member, IEEE*, and Haibing Guan, *Member, IEEE*

***Abstract*—Datacenter networking has gained increasing popularity in the past few years. While researchers paid considerable efforts to enhance the performance and scalability of datacenter networks, achieving reliable data delivery in these emerging networks with misbehaving routers and switches received far less attention. Unfortunately, documented incidents of router compromise underscore that the capability to identify adversarial routers and switches is an imperative and practical need rather than merely a theoretical exercise. To this end, data-plane fault localization (FL) aims to identify faulty links and is an effective means of achieving high network availability. However, existing *secure* FL protocols assume that the source node *knows the entire outgoing path* that delivers the source node's packets and that the path is *static* and *long-lived*. These assumptions are invalidated by the dynamic traffic patterns and agile load balancing commonly seen in modern datacenter networks. We propose the first secure FL protocol, DFL, with *no* requirements on path durability or the source node knowing the outgoing paths. Through a core technique we named delayed function disclosure, DFL incurs little communication overhead and a small, constant router state independent of the network size or the number of flows traversing a router.**

***Index Terms*—Datacenter network, delayed function disclosure, fault localization.**

I. Introduction

As THE infrastructure support for cloud computing, datacenter networks (DCNs) have sparked tremendous interests in the research community, focusing on improving network performance and scalability in *benign* environments with *natural* failures [17]. However, such a generous assumption that switches and routers are trustworthy has proven risky by *real-world* incidents [1], [2], [6], [27], [38], where routers can be compromised due to dishonest employees or social engineering in operational networks and can surreptitiously sabotage network data delivery. These misbehaving routers or switches (or *routers* in general hereinafter) can easily drop, modify, delay, or inject packets in the data plane to mount denial-of-service, surveillance, man-in-the-middle attacks, etc. [32].

Regrettably, current DCNs lack a *secure* way to identify misbehaving routers that jeopardize packet delivery. Existing fault diagnosis approaches in DCNs [26] were designed for nonadversarial settings and are thus vulnerable to attacks. For example, a malicious router can "correctly" respond to `ping` or `traceroute` probes while corrupting other data packets, thus cloaking the attacks from `ping` or `traceroute`. Hence, the increasing demand for high network availability warrants the pursuit of a secure mechanism for identifying malicious routers, referred to as fault localization (FL). FL enables the subsequent investigation, repair, and removal of misbehaving devices, thus benefiting network availability.

Researchers have proposed a wide array of FL protocols for the Internet capable of identifying routers that drop, fabricate, delay, and/or inject packets [8], [9], [12], [40]. However, these protocols were not tailored for DCNs; they require the packet sender to *know the entire packet path* and mandate the path to be *long-lived* (e.g., stable over transmitting 10^8 packets [12]). Recent measurement studies [18], [22] show that a considerable fraction of current DCN flows are short-lived and routing paths are highly dynamic. Furthermore, emerging DCNs call for more agile load balancing and dynamic routing paths. The conflict between the "static-path" assumption and the "dynamic-path" reality renders existing FL protocols inapplicable to DCNs with dynamic traffic patterns and short-lived flows. In addition, existing FL protocols require a router to share cryptographic keys with each source node sending traffic traversing that router, swelling a single router's key storage overhead linear in the number of end nodes; moreover, a router also needs to maintain per-path state for each path traversing that router, making the FL unscalable for today's DCNs that may comprise hundreds of thousands of machines.

We aim to bridge the current gap between the security of FL and the ability to support dynamic traffic patterns in modern DCNs. More specifically, a desired FL protocol should be secure against sophisticated packet dropping, modification, fabrication, and delaying attacks by colluding routers while retaining the following properties.

Manuscript received November 25, 2012; revised May 10, 2013; accepted June 24, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Wang. Date of publication August 28, 2013; date of current version August 14, 2014. This work was supported by CyLab at Carnegie Mellon University, the NSF under Award CNS-1040801, the NSFC under Grant No. 61272101, the Singapore NRF under the CREATE E2S2 Program, and the MOE under Grant No. 313025. This work extends previous work published in the Proceedings of the IEEE Symposium in Security and Privacy, San Francisco, CA, USA, May 20–23, 2012.

X. Zhang is with Datacenter Cluster Management, Google, Pittsburgh, PA 15206 USA (e-mail: xinzhang1228@gmail.com).

F. Zhou, X. Zhu, H. Sun, and H. Guan are with the Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: zhoufanfu@sjtu.edu.cn; zxykobezxy@sjtu.edu.cn; jysunhy@sjtu.edu.cn; hbguan@sjtu.edu.cn).

A. Perrig is with ETH Zurich, Zurich 8092, Switzerland (e-mail: aperrig@inf.ethz.ch).

A. V. Vasilakos is with the University of Western Macedonia, Kozani 50100, Greece (e-mail: vasilako@ath.forthnet.gr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2274662

Path obliviousness: A source node or a router does not need to know the outgoing/downstream path.

Volatile path support: The FL protocol requires no minimum duration time for a forwarding path.

Constant router state: A router does not need to maintain per-path, per-flow, or per-source state.

O(1) key storage: A router only manages a small number of keys regardless of the network size.

Path obliviousness and volatile path support together enable agile load balancing and dynamic routing paths (e.g., valiant load-balanced paths). These two properties also *decouple* the data-plane FL from routing, thus enabling it to support a wide array of routing protocols. Finally, constant router state provides scalability in large-scale networks, and O(1) key storage reduces the security risk due to key compromise.

We observe that the "static-path" assumption in existing secure FL protocols stems from the fact that they operate on entire end-to-end paths (*path-based*) to localize the fault to *one specific link*. Specifically, each router maintains a certain "traffic summary" (e.g., a counter, packet hashes, etc.) for *each* path that traverses the router (thus requiring per-path state), and sends the traffic summary to the source node S of each path. S can then detect a link l as malicious if the traffic summaries from l's two adjacent nodes deviate greatly. Hence, S needs to know the entire path topology to compare traffic summaries of adjacent nodes, and needs to send a large number of packets over the same path so that the deviation in traffic summaries can reflect a statistically accurate estimation of link quality. Finally, to authenticate the communication between the source and each router in the path, a router needs to share a secret key with each source that sends traffic through that router.

In this paper, we explore *neighborhood-based* FL approaches, where a router r's data-plane faults (if any) can be detected by checking the consistency (or conservation) of the traffic summaries generated by the *1-hop neighbors* of r (denoted by $\mathbb{N}(r)$). That is, in benign cases, the packets *sent to* r will be consistent with the packets *received from* r by all of r's neighbors as reflected in their traffic summaries. In this way, the FL is independent of routing paths and only depends on 1-hop neighborhoods, thus supporting arbitrary routing protocols and dynamic load balancing. Additionally, each router in a neighborhood-based approach only needs to maintain state for each neighbor.

Contributions: This paper leverages the neighborhood monitoring approach and delayed function disclosure mechanism to achieve path obliviousness and volatile path support that have not been achieved in other work. Moreover, a router in neighborhood-based FL requires only about 4 MB per-neighbor state, which is 100–10 000 times less than that in path-based FL protocols. Furthermore, this paper emphasizes the importance of secure FL specifically for modern DCNs with dynamic traffic patterns. To the best of our knowledge, DFL is the first approach for achieving secure data delivery in DCNs. In addition, we explore the characteristics of DCN topologies and identify the unique opportunities to optimize the secure fault localization algorithm. More specifically, we devise algorithms for nodes to aggregate local logs in the monitoring protocol to a controller node with negligible communication overhead. We present both theoretical analysis and experimental results of the proposed algorithms and demonstrate the resulting communication overhead is 10 times lower than that in DynaFL [41], thus making the neighborhood-based FL truly secure *and* practical. We anticipate that work will spark future research endeavors in tackling network-level security challenges in datacenter networks and the neighborhood-based FL to serve as a building block to achieve high availability for other networked systems.

II. Problem Statement

A. Assumptions

We consider a network with dynamic traffic patterns and a relatively static network topology, which is best exemplified by today's datacenter networks. To provide maximum flexibility to support various routing protocols, and even packet-level load balancing, we pose *no* restriction on the routing protocols and load balancing mechanisms used in the network. We do not make any assumption on the network topology (e.g., tree-based, etc.) for the sake of wide applicability of the FL protocol, though for a particular type of topology, potential optimizations may exist. We assume a *trusted administrative controller* (AC) in the network, which shares a pairwise secret key with each node in the network. As we will show later, the AC is mainly in charge of analyzing the traffic summaries gathered from different nodes and localizing any neighborhood with data-plane faults. We argue it is feasible to deploy an AC for a datacenter network (as in 4D [21], SANE [15], OpenFlow [31], etc.), which is usually under a single administrative domain. The AC is a logically single entity, but can be implemented in a distributed way. Finally, we require nodes in the network to be *loosely* time-synchronized, e.g., the order of milliseconds.

B. Adversary Model

We consider a sophisticated adversary controlling multiple malicious nodes. Specifically, a *malicious node* corrupts data-plane packets by unexpectedly *dropping*, *modifying*, and *delaying* legitimate packets sent by the source and *fabricating* bogus packets that are not sent by the source. A malicious node can corrupt both the data packets and *control packets*, such as traffic summaries sent from each node to the AC and certain administrative messages sent from the AC to other nodes. Furthermore, a sophisticated adversary has knowledge of and tries to game the FL protocol to evade detection. Multiple *colluding nodes* can collectively perform the above data-plane attacks, conspiring to evade detection or frame benign nodes. The colluding nodes know each other's security credentials (e.g., secret keys used in the FL protocol).

C. Problem Formulation

Our goal is to design a practical and secure *neighborhood-based* FL protocol to identify a suspicious *neighborhood* (if any) that contains at least one malicious node. Recall that practicality translates to *path obliviousness*, *volatile path support*, and *constant router state* as stated in Section I. We further adopt the (α, β, δ) *-accuracy* [20] to formalize the security requirements as follows.

- If more than β fraction of the packets are corrupted by a malicious node m, the FL protocol will raise a neighborhood containing m *or* one of its colluding nodes as suspicious with probability at least $1 - \delta$.

- In benign cases, if no more than α fraction of the packets are spontaneously corrupted (e.g., dropped) in a neighborhood, the FL protocol will raise the neighborhood as suspicious with probability at most δ.

The thresholds α and β are introduced to tolerate spontaneous failures (e.g., natural packet loss) and are set by the network administrator based on her experience and expectation of network performance. Although false positives (claiming a benign neighborhood as malicious) caused by such failures and different thresholds will waste some dedicated overhead for investigation, they are not a big concern as long as the false positive rate is low. Additionally, the false negative rate remains low if the thresholds are set properly (in Section VIII). Moreover, the upper-layer protocol such as TCP can retransmit these packet losses.

Neighborhood-based FL enables the network administrator to scope further investigation to a 1-hop neighborhood to find out which router is compromised. It is also possible to further employ dedicated monitoring protocols, which only need to monitor a small region (the identified neighborhood) of the network to find the specific misbehaving router.

III. Challenges and Overview

In this section, we sketch a general neighborhood-based FL protocol and highlight the security and scalability challenges. We then present the key ideas in DFL that answer these challenges. We start with the notations used throughout the paper.

Notation: We use the terms *node* and *router* interchangeably to generally refer to devices that either perform layer-2 switching or layer-3 routing. We denote the 1-hop neighborhood of a node s as $\mathbb{N}(s)$. For a particular packet traversing a neighborhood $\mathbb{N}(s)$, the neighbor sending that packet to node s is called an *ingress node* in $\mathbb{N}(s)$ for that packet, and the node receiving that packet *from* s is called an *egress node*. We term a sequence of packets as a *packet stream* $\mathbb{S}$. Particularly, we denote the packet stream sent from node i to node j as $\mathbb{S}_{ij}$, and this packet stream is *seen* by nodes i and j as $\mathbb{S}_i^{\rightarrow j}$ and $\mathbb{S}_j^{\leftarrow i}$, respectively. The *difference* of two packet streams $\mathbb{S}$ and $\mathbb{S}'$, denoted by $\Delta(\mathbb{S}, \mathbb{S}')$, refers to the number of packets in one packet stream but not in the other, without considering the variant IP header fields such as the time to live (TTL) and checksum.

A. High-Level Steps

The general steps in neighborhood-based FL are: 1) *recording* local traffic summaries; 2) *reporting* the traffic summaries to the AC; and 3) *detecting* suspicious neighborhoods by the AC based on the received traffic summaries, as we sketch in the following.

Recording: We divide the time in a network into consecutive *epochs*, which are synchronous among all the nodes including the AC in the network. For each neighbor r, a node s locally generates traffic summaries, denoted by $TS_s^{\rightarrow r}$ and $TS_s^{\leftarrow r}$, for the packet streams $\mathbb{S}_{sr}$ and $\mathbb{S}_{rs}$ in each epoch, respectively. Fig. 1 depicts the router state in a toy example.

The traffic summary recorded by a node s should reflect both the packet contents and the arrival/departure time seen at node s to enable the detection of malicious packet corruption and delay. Our FL protocol is based on the assumption that the packets sent to a node should be consistent with the packets received from the node in the 1-hop neighbor. The source or destination node only generates or receives the packets, so the packets from the source or destination in each node should be excluded from consistency check. Fortunately, each node can tell the source and destination from the packet header, and can then exclude these packets in packet streams from the counting (e.g., $\mathbb{S}_{sr}$ and $\mathbb{S}_{sr}$). For the sake of scalability, the traffic summary cannot simply be an entire copy of all the original packets (or their hashes using a *cryptographic* hash function such as SHA-1 that provides one-wayness and collusion resistance) and their timing information. Instead, we use a *fingerprinting function* $\mathcal{F}$ to reflect the *aggregates* of packet contents to save both the router state and bandwidth consumption for reporting the traffic summaries to the AC. We denote the fingerprint for a packet stream $\mathbb{S}_{rs}$ generated by r as $\mathcal{F}(\mathbb{S}_r^{\rightarrow s})$, as Fig. 1 depicts. In addition, as Fig. 1 shows, for a packet stream $\mathbb{S}_{rs}$ (or $\mathbb{S}_{sr}$), the traffic summary of node r also contains the *average* departure time $\overline{t}_r^{\rightarrow s}$ (or arrival time $\overline{t}_r^{\leftarrow s}$) and the total number of packets $n_r^{\rightarrow s}$ (or $n_r^{\leftarrow s}$) in $\mathbb{S}_{rs}$ (or $\mathbb{S}_{sr}$) seen in the current epoch to enable the detection of packet delay attacks.

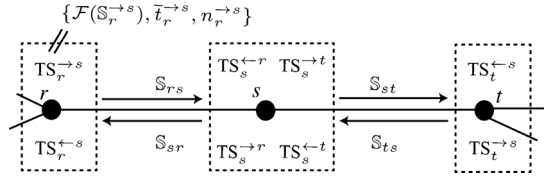


Fig. 1. Router state for traffic summaries.

Reporting: At the end of each epoch, each node s sends its local traffic summaries to the AC .

Detection: After receiving the traffic summaries at the end of an epoch, the AC runs a consistency check over the traffic summaries in each neighborhood. A large inconsistency of the traffic summaries in a certain neighborhood $\mathbb{N}(s)$ indicates that $\mathbb{N}(s)$ is suspicious.

B. Fingerprinting Function $\mathcal{F}$

Before we present the instantiation of $\mathcal{F}$, we first describe the general properties that $\mathcal{F}$ should satisfy. To enable the AC to detect suspicious neighborhoods, $\mathcal{F}$ should generate traffic summaries with the following two properties:

Property 1: Given any two packet streams $\mathbb{S}$ and $\mathbb{S}'$, the "difference" between $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$ can give an estimation of the difference between $\mathbb{S}$ and $\mathbb{S}'$, denoted by: $\Delta(\mathcal{F}(\mathbb{S}), \mathcal{F}(\mathbb{S})') \rightsquigarrow \Delta(\mathbb{S}, \mathbb{S}')$.

Defining the "difference" between $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$ is $\mathcal{F}$-specific, as we show shortly.

Property 2: Given any two packet streams $\mathbb{S}$ and $\mathbb{S}'$, $\mathcal{F}(\mathbb{S} \cup \mathbb{S}') = \mathcal{F}(\mathbb{S}) \cup \mathcal{F}(\mathbb{S}')$.

The $\cup$ operator on the left-hand side denotes a union operation of the two packet streams $\mathbb{S}$ and $\mathbb{S}'$. The $\cup$ operator on the right-hand side denotes a "combination" of $\mathcal{F}(\mathbb{S})$ and $\mathcal{F}(\mathbb{S}')$, which is $\mathcal{F}$-specific and defined shortly.

These two properties enable the conversion *from checking packet stream conservation to checking the conservation of traffic summaries in a neighborhood*. In other words, these two properties enable nodes to simply store the compact packet fingerprints instead of the original packet streams while still enabling the AC to detect the number of packets dropped, modified, and fabricated between two packet streams from their corresponding fingerprints.

During the detection phase, the AC only needs to compare the difference between: 1) the *combined* traffic summaries for packets *sent to* node s in $\mathbb{N}(s)$, i.e., $\cup_{i\in\mathbb{N}(s)}\mathcal{F}(\$_i^{\rightarrow s})$; and 2) the *combined* traffic summaries for packets *received from* node s in $\mathbb{N}(s)$, i.e., $\cup_{i\in\mathbb{N}(s)}\mathcal{F}(\$_i^{\leftarrow s})$. By Properties 1 and 2

$$\begin{aligned}&\Delta\Big(\underset{i\in\mathbb{N}(s)}{\cup}\mathcal{F}(\$_i^{\rightarrow s}),\underset{i\in\mathbb{N}(s)}{\cup}\mathcal{F}(\$_i^{\leftarrow s})\Big)\\&=\Delta\Big(\mathcal{F}\Big(\underset{i\in\mathbb{N}(s)}{\cup}\$_i^{\rightarrow s}\Big),\mathcal{F}\Big(\underset{i\in\mathbb{N}(s)}{\cup}\$_i^{\leftarrow s}\Big)\Big)\ \text{based on Property 2}\\&\rightsquigarrow\Delta\Big(\underset{i\in\mathbb{N}(s)}{\cup}\$_i^{\rightarrow s},\underset{i\in\mathbb{N}(s)}{\cup}\$_i^{\leftarrow s}\Big)\ \text{based on Property 1.}\quad(1)\end{aligned}$$

Note that $\Delta(\cup_{i\in\mathbb{N}(s)}\$_i^{\rightarrow s},\cup_{i\in\mathbb{N}(s)}\$_i^{\leftarrow s})$ reflects the discrepancy between packets sent to and received from node s, and a large discrepancy indicates packet dropping, modification, and fabrication attacks in $\mathbb{N}(s)$.

Sketch for $\mathcal{F}$: The pth moment estimation sketch [4], [16], [39] (as used by Goldberg *et al.* [20] for *path-based* FL) serves as a good candidate for $\mathcal{F}$. More specifically, pth moment estimation schemes use a random linear map to transform a packet stream into a short vector, called the sketch, as the traffic summary. In *benign* cases, packets, if viewed as 1.5-kB (the Maximum Transmission Unit) bit-vectors, are "randomly" drawn from $\{0,1\}^{1536\times 8}$. Hence, different packet streams will result in different sketches *with a very high probability (w.h.p.)*. Goldberg *et al.* [20] also extensively studied how to estimate the number of packets dropped, injected, or modified between two packet streams from the "difference" of two corresponding sketch vectors, thus satisfying Property 1. We also previously proved that the sketch satisfies Property 2 [41]. Specifically, the difference $\Delta(\mathcal{F}(\$),\mathcal{F}(\$)')$ (used in Property 1) between two sketches is defined as

$$\Delta(\mathcal{F}(\$),\mathcal{F}(\$)')=\|\mathcal{F}(\$)-\mathcal{F}(\$)'\|_p^p \quad (2)$$

where $\|x\|_p^p$ denotes the pth moment of the vector x. The combination of $\mathcal{F}(\$)$ and $\mathcal{F}(\$)'$ used in Property 2 is defined as

$$\mathcal{F}(\$)\cup\mathcal{F}(\$)'=\mathcal{F}(\$)+\mathcal{F}(\$)' \quad (3)$$

where + denotes the addition of two vectors.

C. Challenges in a Neighborhood-Based Fl

From Property 1, we can further derive the following conditions on the fingerprinting function $\mathcal{F}$. Given any two packet streams $\$_r$ and $\$_t$ seen at nodes r and t, respectively, a fingerprinting function computed by r and t should satisfy

$$\text{if } \$_r=\$_t,\mathcal{F}(\$_r)=\mathcal{F}(\$_t) \quad (4)$$

$$\text{if } \$_r\neq\$_t,\mathcal{F}(\$_r)\neq\mathcal{F}(\$_t)\ \text{w.h.p.} \quad (5)$$

The first condition ensures the consistency of traffic summaries (more precisely, sketches in the traffic summaries) in the benign case when the packet streams are not corrupted. The second condition ensures that if packet corruption happens between nodes r and t, inconsistency of the traffic summaries will be observed, which will then enable the estimation of packet difference in the corresponding packet streams (Property 1). However, these two conditions tend to be contradicting with the following dilemma.

$\mathcal{F}$ Without Different Secrets: If the random linear map in $\mathcal{F}$ (which can be implemented as a hash function [12]) is *not* computed with different secret keys by different nodes, a malicious node can predict the $\mathcal{F}$ output of *any other* node for *any* packet. Since $\mathcal{F}$ maps a set of packets (or their 160-bit cryptographic hashes) to a much smaller sketch, *hash collisions* will exist where two different packets produce the same $\mathcal{F}$ output (since sketch is not proven to preserve the collision resistance property of the cryptographic hash function). Hence, a malicious node can leverage such collisions to modify packets such that the modified/fabricated packets will produce the same $\mathcal{F}$ output at other nodes, violating the condition in (5). Fig. 2 depicts such an example.

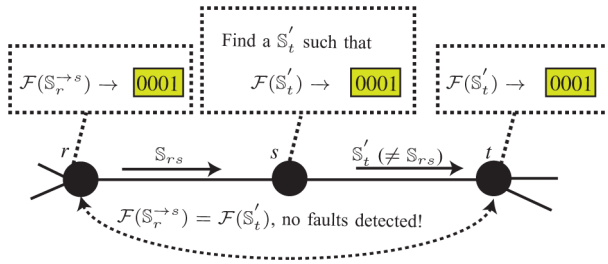


Fig. 2. Example of stealthy packet modification attacks when nodes do not use different secret keys for computing $\mathcal{F}$. For simplicity, the sketch vector is represented as a "0–1" bit vector. The malicious node s modifies the packet stream in such a way that the modified packet stream $\mathbb{S}_t'$ still results in the same sketch vector as $\mathbb{S}_{rs}$ at node t.

$\mathcal{F}$ With Different Secrets: If nodes compute $\mathcal{F}$ with different secret keys to satisfy the condition in (5), it is hard for the AC to perform a consistency check among the resulting sketches. For example, even the same packet stream would result in different sketches at different nodes, thus violating the condition in (4). Since the sketch is only a *compact and approximate* representation of the original packet stream, the AC cannot revert the received sketches to the original packet streams to check packet stream conservation.

Scalability Versus Sampling: Even with $\mathcal{F}$ for packet fingerprinting, a traffic summary over a huge number of packets can become too bandwidth-consuming to be sent frequently to the AC (e.g., every 20 ms). For example, the number of packets for an OC-192 link (10 Gb/s) can be on the order of 10^7 per second in the worst case, which swells the size of a sketch to hundreds of bytes to bound the false positive rate below 0.001 [20] and may require several kB/s bandwidth for the reporting by *each* node. Packet sampling represents a popular approach to reducing bandwidth consumption, where each node only samples a *subset* of packets to feed into $\mathcal{F}$ for generating the traffic summaries. To enable a consistency check of the traffic summaries in a neighborhood, all nodes in a neighborhood should sample the *same* subset of packets, and the challenge is how to efficiently decide which subset of packets all nodes should agree to sample. For security, the sampling scheme must ensure that a malicious node cannot predict whether a packet to be forwarded will be sampled or not. Otherwise, the malicious node can drop any nonsampled packets without being detected.

The problem is further complicated by the presence of *collusion attacks* in our strong adversary model as well as our *path obliviousness* requirement. Several existing sampling schemes are broken when applied to our setting. For example, in Symmetric Secure Sampling (SSS) [20], the packet sender and receiver use a shared Pseudo-Random Function (PRF) $\mathcal{P}$ to coordinate their sampling. Imported to our setting, e.g., using the

neighborhood example in Fig. 2, nodes r and t share a secret key K_{rt} and a PRF $\mathcal{P}$, compute $\mathcal{P}$ with K_{rt} for each packet, and sample the packet if the PRF output is within a certain range. In this way, node s *itself* cannot know whether a packet is sampled or not. However, this approach fails in our setting. For example in Fig. 2, we have the following.

- If s and r collude, r can inform s of which packets are sampled, so that s can safely drop nonsampled packets and not be detected.
- Due to the dynamic traffic pattern, an ingress node r of a neighborhood $\mathbb{N}(s)$ does not know which egress node a packet will traverse in $\mathbb{N}(s)$ (if s has more neighbors than r and t, there exist multiple possible egress nodes than t). Hence, r does not know which PRF or secret key to use for packet sampling, given that r shares a different secret key with each node in $\mathbb{N}(s)$.

So even when the majority of a benign node's neighborhood are compromised, the AC can figure out the inconsistency by inspecting the report from the benign node.

D. DFL Key Ideas

In DFL, nodes temporarily store the cryptographic hashes (which are collision-resistant) for all packets received/sent *per neighbor* in an epoch. At the end of each epoch e, nodes use *epoch sampling* to decide if packets in the epoch are to be fingerprinted; if so, nodes generate the traffic summaries and report them to the AC. This reduces both the communication overhead for sending the traffic summaries to the AC and the computational overhead for generating and checking the traffic summaries. Specifically, nodes first use the *same* per-epoch *sampling key* K_{s}^e (described shortly) for computing a PRF $\mathcal{P}$ to determine if the current epoch is "selected"; if and only if the current epoch is selected, nodes will use $\mathcal{F}$ with the *same* per-epoch *fingerprinting key* K_{f}^e (described shortly) to map packets into per-neighbor traffic summaries. Using the same K_{s}^e and K_{f}^e enables consistency checking over the traffic summaries from different nodes.

To address the packet modification attacks and collusion attacks mentioned earlier, nodes do *not* know the per-epoch K_{s}^e and K_{f}^e until the *end* of each epoch e, after they *have forwarded* (or possibly corrupted) packets in epoch e. Thus, when a packet is to be forwarded (or corrupted), a malicious node does not know K_{s}^e and K_{f}^e, and thus cannot predict whether this epoch is selected for sending traffic summaries, and if selected, what the sketch output will be for this packet. To achieve this property, in DFL, the trusted AC periodically sends the per-epoch K_{s}^e and K_{f}^e via *function disclosure messages* to all nodes at the end of each epoch in a reliable way (described later), and nodes use the received K_{s}^e and K_{f}^e to select epochs and fingerprint packets that have already been forwarded or corrupted.

A malicious node may first attempt to locally hold all the packets in an epoch e, and only forward or corrupt packets at the end of e when the malicious node learns K_{s}^e and K_{f}^e, thus being able to launch the sophisticated packet modification and selective packet corruption attacks as mentioned earlier. However, since the traffic summaries also include the average departure/arrival time of the sent/received packets, the malicious node will be detected with packet delay misbehavior in the detection phase.

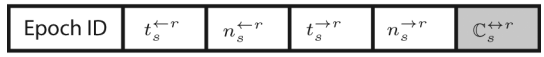


Fig. 3. Router per-neighbor state details.

IV. Recording Traffic Summaries

The technical challenges in the recording phase are how to deal with *imperfect* time synchronization among nodes and packet transmission delay, and how to efficiently protect the function disclosure message from adversarial corruption. We explain how DFL solves these challenges in turn.

A. Storing Packets

In the "ideal" case (with perfect time synchronization and no packet transmission delay), nodes simply need to store packets for the single "current" epoch and, at the end of each epoch, send the traffic summaries to the AC for that epoch. However, in practice, routers need to determine to which epoch an incoming packet belongs (or whether a received packet belongs to the current epoch or a previous, outdated epoch). One might attempt to let routers map received packets into epochs based on their local packet arrival time. However, this approach would introduce large errors (because of misaligned time or network transmission delay).

To deal with imperfect time synchronization, the source in DFL embeds a *local* timestamp when sending each packet. Such a timestamp can be added as an additional flow header, using the TCP timestamp, or in the IP option field that all routers can process efficiently. Any router in the forwarding path will determine the corresponding epoch for each packet based on the embedded timestamp. In this way, we ensure that all routers put each packet in the same epoch for updating the traffic summaries. For example, if the timestamp embedded by the source is t_s and the epoch length is L, then all routers will map the packet into epoch $\lfloor \frac{t_s}{L} \rfloor$.

To eliminate traffic summary inconsistencies due to packet transmission delay, we also need to ensure that when generating traffic summaries for a certain epoch e, packets that are sent and not corrupted in epoch e are received by *all* the nodes in the forwarding paths. To this end, if the epoch length is set to L and the expected upper bound on the *one-way* packet transmission delay in the network is D, each router stores packets sent in the current epoch e *as well as* in previous $\lceil \frac{D}{L} \rceil$ epochs, denoted by $e-1, e-2, \ldots, e-\lceil \frac{D}{L} \rceil$. We call these epochs *live epochs*. Then, at the end of an epoch e, nodes will generate and send to the AC the traffic summaries for the *oldest* live epoch $e-\lceil \frac{D}{L} \rceil$, in which the packets have either traversed all nodes in their forwarding paths or been corrupted. The periodic function disclosure messages that the AC sends synchronize the current epoch ID and the oldest live epoch ID for which traffic summaries are needed for reporting.

Hence, a node s maintains the following data structures for each neighbor r for each epoch, as Fig. 3 also shows.

- The packet cache $\mathbb{C}_s^{\leftrightarrow r}$ temporarily stores hashes for packets in both $\mathbb{S}_s^{\rightarrow r}$ and $\mathbb{S}_s^{\leftarrow r}$ that are seen in a live epoch (using a cryptographic hash function such as SHA-1). Each entry contains the packet hash and a bit indicating if the packet belongs to $\mathbb{S}_s^{\rightarrow r}$ or $\mathbb{S}_s^{\leftarrow r}$.

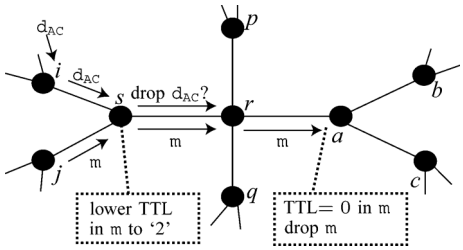


Fig. 4. Possible attacks in the recording phase. A malicious node s may attempt to drop the function disclosure message d_{AC}, or manipulate the TTL value to cause packets to be dropped at a remote place (node a in this example), thus framing a remote neighborhood ($\mathbb{N}(a)$ in this example).

- The router stores the *sum* of packet departure timestamps $t_s^{\rightarrow r}$ seen in $\mathbb{S}_s^{\rightarrow r}$ and the sum of packet arrival timestamps $t_s^{\leftarrow r}$ seen in $\mathbb{S}_s^{\leftarrow r}$ in a live epoch with millisecond precision.
- Finally, the router stores the total number of packets $n_s^{\rightarrow r}$ seen in $\mathbb{S}_s^{\rightarrow r}$ and $n_s^{\leftarrow r}$ seen in $\mathbb{S}_s^{\leftarrow r}$ in a live epoch.

Among these data structures, $t_s^{\leftarrow r}$, $t_s^{\rightarrow r}$, $n_s^{\leftarrow r}$, and $n_s^{\rightarrow r}$ require small constant storage, around 8 or 4 B for each. $\mathbb{C}_s^{\leftrightarrow r}$ will be used for packet fingerprinting. The size of $\mathbb{C}_s^{\leftrightarrow r}$ depends only on the epoch length L and link bandwidth, but not the number of flows/paths traversing node s. As Section VIII-A shows, with an epoch length of 20 ms and one-way network latency of 20 ms, each router line-card requires only around 4 MB of memory for an OC-192 link, which is readily available today.

For the sake of simplicity, we use $\mathbb{C}_s^{\rightarrow r}$ and $\mathbb{C}_s^{\leftarrow r}$ to denote the packets cached for $\mathbb{S}_s^{\rightarrow r}$ and $\mathbb{S}_s^{\leftarrow r}$ by node s, respectively.

B. Secure Function Disclosure

At the end of each epoch e, the AC discloses the sampling key $K_{\mathrm{s}}^{e-\lceil \frac{D}{L} \rceil}$ and fingerprinting key $K_{\mathrm{f}}^{e-\lceil \frac{D}{L} \rceil}$ to all nodes in the network via a *function disclosure message* d_{AC}, and requests the traffic summaries for the oldest live epoch $e - \lceil \frac{D}{L} \rceil$. Obviously, d_{AC} itself needs to be protected from data-plane attacks (dropping, modification, fabrication, or delaying) by a malicious node during end-of-epoch broadcasting. It might be tempting to let the AC use digital signatures to authenticate d_{AC} in order to address malicious modification and fabrication. However, frequently generating and verifying the signatures on a per-epoch basis can be expensive (e.g., an epoch can be as short as 20 ms, and signature generation and verification time could be on the order of milliseconds).

Fortunately, the function disclosure message d_{AC} is transmitted at the end of each epoch synchronously among all the nodes. If a malicious node s drops d_{AC}, the AC will fail to receive the traffic summaries of certain neighbors of s, thus detecting $\mathbb{N}(s)$ as suspicious. For example in Fig. 4, if s drops d_{AC} instead of forwarding it to its neighbor r, node r cannot fingerprint the packets to generate traffic summaries, thus failing the consistency check of traffic summaries in $\mathbb{N}(s)$. As we show in Section V, the AC expects to receive traffic summaries within a short amount of time after each epoch ends; delaying d_{AC} more than that amount of time is effectively equivalent to dropping d_{AC} and causes the malicious node's neighborhood to be detected. Thus, the remaining problem is to prevent the modification and fabrication of d_{AC}, which is equivalent to authenticating d_{AC} to all nodes in the network *without* the use of digital signatures. Section VII further elaborates why the authentication of d_{AC} is needed for security purposes.

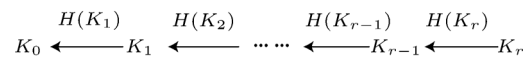


Fig. 5. One-way hash chain example.

In DFL, time in the network is loosely time-synchronized and divided into consecutive epochs; the authentication of d_{AC} is required only once per epoch. This setting is naturally aligned with that of the TESLA broadcast authentication [36], which authenticates broadcast messages (d_{AC} in our case) using only Message Authentication Codes (MACs) with keys derived from a one-way hash chain. As Fig. 5 shows, the AC applies a one-way hash function H repeatedly on the root key K_{r} to derive a set of epoch authentication keys, and uses key K_e to compute a MAC for authenticating d_{AC} in epoch e. The AC publishes K_0 through the network so that nodes can verify if any given epoch key is indeed derived from the genuine one-way hash chain. Then, d_{AC} in epoch e includes: 1) the current epoch ID e, the oldest live epoch ID $j = e - \lceil \frac{D}{L} \rceil$ to be examined, sampling and fingerprinting keys, a MAC computed with K_e for the current epoch; and 2) the key K_j for computing the MAC in a *previous* epoch j, by which nodes can verify the authenticity of d_{AC} in epoch j (verification delayed by $\lceil \frac{D}{L} \rceil$ epochs), i.e.,

$$\mathsf{d}_{\mathsf{AC}} = e \| j \| K_{\mathrm{s}}^j \| K_{\mathrm{f}}^j \| \mathrm{MAC}_{K_e}(e \| j \| K_{\mathrm{s}}^j \| K_{\mathrm{f}}^j) \| K_j \qquad (6)$$

where $\|$ denotes concatenation. Section VII describes the reason for disclosing the key for epoch $j = e - \lceil \frac{D}{L} \rceil$ instead of epoch $e - 1$.

Furthermore, DFL creates a spanning tree in the network rooted at the AC, along which d_{AC} is delivered to each node. Since DFL uses a *pregenerated, static* spanning tree for the broadcast messages, there is no need for dynamic path support when protecting d_{AC} .

C. Sampling and Fingerprinting

Given the disclosed K_{s}^j and K_{f}^j at the end of an epoch e, each node t first uses the sampling PRF $\mathcal{P}$ with K_{s}^j, denoted by $\mathcal{P}_{K_{\mathrm{s}}^j}$, to determine if the oldest live epoch j is selected. If so, node t then uses the fingerprinting function $\mathcal{F}$ to map the cached packet hashes in each per-neighbor stream into a sketch vector, i.e., $\mathcal{F}_{K_{\mathrm{f}}^j}(\mathbb{C}_t^{\rightarrow r})$ or $\mathcal{F}_{K_{\mathrm{f}}^j}(\mathbb{C}_t^{\leftarrow r})$, computed with the given K_{f}^j. Finally, node t generates *two* traffic summaries $T_t^{\rightarrow r}$ and $T_t^{\leftarrow r}$ for a neighbor r for packet streams $\mathbb{S}_t^{\rightarrow r}$ and $\mathbb{S}_t^{\leftarrow r}$, respectively.

- $T_t^{\rightarrow r}$ includes a fingerprint $\mathcal{F}_{K_{\mathrm{f}}^j}(\mathbb{C}_t^{\rightarrow r})$, average packet departure time $\overline{t}_t^{\rightarrow r} = \frac{t_t^{\rightarrow r}}{n_t^{\rightarrow r}}$, and the total number $n_t^{\rightarrow r}$ of packets seen in $\mathbb{S}_t^{\rightarrow r}$ in epoch j.
- $T_t^{\leftarrow r}$ includes a fingerprint $\mathcal{F}_{K_{\mathrm{f}}^j}(\mathbb{C}_t^{\leftarrow r})$, average packet arrival time $\overline{t}_t^{\leftarrow r} = \frac{t_t^{\leftarrow r}}{n_t^{\leftarrow r}}$, and the total number $n_t^{\leftarrow r}$ of packets seen in $\mathbb{S}_t^{\leftarrow r}$ in epoch j.

Implementing $\mathcal{P}$: Specifically, $\mathcal{P}$ maps an epoch ID to an n-bit integer uniformly distributed in $[0, 2^n - 1]$. Given a *sampling rate* $\lambda \in (0, 1)$, a node computes $\mathcal{P}_{K_{\mathrm{s}}^j}$ over the epoch ID j that is being examined, and epoch j is selected iff $\mathcal{P}_{K_{\mathrm{s}}^j}(j) < \lambda \cdot 2^n$. In this way, on average a fraction λ of the epochs will be selected. Since nodes use $\mathcal{P}$ with the same K_{s}^j for epoch

sampling, in the benign case, nodes will select the same set of epochs, thus ensuring the consistency of the traffic summaries in a neighborhood.

Implementing $\mathcal{F}$: We use the second-moment sketch computed with K_{f}^{j} as a case study to implement $\mathcal{F}$, and analyze the size of the sketch vector to achieve Property 1 with the (α, β, δ)-accuracy. We assume 10^7 packets per second in the worst case for an OC-192 link with an epoch length of L (seconds). Then, the number of packets η in a sampled epoch is $\eta = L \cdot 10^7$. Using the classical Sketch due to Alon *et al.* [5] for example, the storage requirement for the sketch is given by

$$M \times \log_2 \sqrt{2\eta \ln\left(\frac{200N}{\delta}\right)} \tag{7}$$

where

$$M > \frac{12}{\epsilon^2}\frac{1}{3-2\epsilon}\ln\frac{1}{\delta} \quad \text{and} \quad \epsilon = \frac{\beta-\alpha}{\beta+\alpha}. \tag{8}$$

In Section VIII-A, we derive numeric values for the size of the sketch vector based on the epoch length L.

Dealing With TTL Attacks: Certain fields in the IP header, such as the TTL, checksum, and some IP option fields, will change at each hop. Both sampling and fingerprinting in DFL need to properly deal with these variant fields to avoid both false positives and false negatives. Take the TTL field for instance hereinafter (though the arguments apply similarly to other variant fields). On the one hand, if $\mathcal{P}$ and $\mathcal{F}$ are computed over the entire packets including the TTL field, even in the benign case the same packet stream will leave different traffic summaries (or precisely, the sketch vectors) at ingress and egress nodes. On the other hand, if $\mathcal{P}$ and $\mathcal{F}$ are computed over the entire packets excluding the TTL field, a malicious node can modify the TTL field at liberty without affecting the traffic summaries. Fig. 4 depicts an example TTL attack, where the malicious node s lowers the TTL value to 2 in the packets and causes the packets to be dropped at the 2-hop-away downstream node a, thus framing neighborhood $\mathbb{N}(a)$.

To address the TTL attacks, when computing $\mathcal{P}$ and $\mathcal{F}$, each node r performs either of the following.

- For a packet received from a neighbor, node r computes $\mathcal{P}$ and $\mathcal{F}$ over the packet *including* the TTL.
- For a packet sent to a neighbor, node r computes $\mathcal{P}$ and $\mathcal{F}$ over the packet, but with the TTL field additionally decreased by 2 (equal to the TTL value at the 2-hop-away egress node in $\mathbb{N}(r)$).

In this way, node r in Fig. 4 simply uses the TTL value as contained in the packets received from s when computing $\mathcal{F}$ and $\mathcal{P}$ since the ingress nodes in $\mathbb{N}(s)$ (nodes i and j) must have computed $\mathcal{F}$ and $\mathcal{P}$ with an adjusted TTL value equal to that at node r.

The TTL value in a packet is also decremented by one for every second the packet is buffered at a router. Holding a packet longer than 1 s at a router is treated as a packet delaying attack and will be detected due to the use of the above construction.

V. Reporting Traffic Summaries

Reporting traffic summaries suffer both security and scalability challenges as explained in turn in this section.

Security: In DFL, all servers periodically send traffic summaries to the AC along their reporting paths. We need to prevent these traffic summaries from being dropped, modified, or injected along the reporting paths. To this end, DFL utilizes an *Onion Authentication* approach [40], [43] to protect the transmission of both the traffic summary reports and d_{AC} along each path.

Scalability: During the reporting phase, the huge volume of reporting traffic summaries from *all* nodes to the AC may easily cause undesired link load and congestion in some nodes in the network. Given that modern datacenters can easily contain up to hundreds of thousands of nodes, the link with heaviest reporting load may become the bottleneck and hamper the entire DCN scalability. For example, DynaFL simply employs a Minimum Spanning Tree for each node to send local traffic summaries to the AC; as a result, the reporting alone may consume more than 10-MB/s bandwidth on a certain link if no epoch sampling is used [41]. In this section, we design two algorithms for addressing the scalability challenge of the reporting phase taking into account computational overhead for deriving the reporting paths and link congestion due to the transmission of the traffic summaries. One algorithm, called LevAdj, uses level adjustment-based tree, and each node utilizes exactly one path to the AC for reporting, while the other algorithm is called BinDin, which combines the Dinic flow algorithm with binary search and is based on multipath reporting.

The primary motivation for designing two algorithms is to accommodate two different network topologies: "symmetrically structured" topologies specifically for DCNs and general graph topologies for generic, abstract network model. The characteristics of "symmetrically structured" topologies are regular symmetric and multihierarchical, and where a source node A has multiple equal-length paths to any end node B [17], while we do not require any layering or pattern for the general graph topologies. Based on these models, we design the LevAdj algorithm for the DCN topologies and the BinDin algorithm for more general network topologies for the sake of completeness and comparison. The results in Section VIII-C show that there is little difference regarding the bandwidth overhead between the two algorithms in DCNs. Thus, the LevAdj algorithm is well suited for DCN topologies since it also incurs less computational overhead. The BinDin algorithm consumes less bandwidth overhead in general topologies, but requires more computational time than the LevAdj algorithm. Given each algorithm has its merits, we present the two algorithms in this paper with a discussion of their tradeoffs.

In the following Sections V-A and V-B, we describe our two algorithms for alleviating link load due to reporting traffic summaries. We first propose the LevAdj algorithm for DCN topologies and then present the BinDin algorithm for general network topologies.

A. LevAdj Algorithm

Considering that the network topology in data center networks tends to be strictly hierarchical with a strict layering pattern [17], it is natural and promising for us to make use of a tree for the reporting phase. Generating a spanning tree based on the network topology structure is simple and fast, and it is possible for us to dynamically change the report spanning tree after detecting that one or more nodes were compromised or crashed.

The key challenge is to avoid creating a hotspot link in the spanning tree. However, in the following we prove that generating a tree that minimizes the reporting load on the busiest link is NP-complete.

Complexity Proof: Suppose we make use of a tree as the reporting paths for all nodes, i.e., all the reporting messages are transmitted along a spanning tree rooted at AC. Minimizing the reporting load in the busiest link is equivalent to getting the minimum congestion tree based on the network topology graph. Let G be a network topology graph and T be a spanning tree of G. The congestion of G in T, denoted by $cng_G(T)$, is the maximal congestion over all the edges in T. To prevent network congestion, we should minimize the load of the busiest link in the spanning tree, that is, to find the spanning tree T whose congestion is minimum in graph G. From the relation between the "optimization problem" and "decision problem," finding the minimum congestion spanning tree in the graph can be reduced to deciding whether there exists a spanning tree T in graph G whose congestion does not exceed a certain amount k (some integer). According to a proven theorem [34], it is NP-complete to decide $cng_G < k$ for planar graph G and integer k. Hence, obtaining the minimum busiest link load based on a tree topology structure is NP-complete.

Therefore, it is impractical for us to get the optimal solution, and we design LevAdj as a heuristic to efficiently obtain a good spanning tree. In the following, we provide the LevAdj algorithm details.

Algorithm Details: In LevAdj, we require the reporting paths to form a tree for simplicity, i.e., we do not consider multipath as the reporting topology. In addition, we let every node only choose the shortest path to the AC (root of the tree). These restrictions can simplify the problem while at the same time not degrading the performance too much. Based on these restrictions, nodes can be divided into different groups according to its distance to the AC node. We use G_i to represent the set of nodes whose distance to the AC node is i (hops), and the parent of a node in the reporting tree in G_i must belong to G_{i-1}. In this case, the reporting load of node n in group G_i is just the reporting load on the edge between node n and its parent node. Thus, the problem of alleviating the reporting load of the heaviest edge can be reduced to the problem of reducing the load of the busiest node.

The LevAdj algorithm aims to generate the tree-based paths for sending traffic summaries and consists of two phases, i.e., the initialization of the tree and the adjustment of the tree. To initialize the tree, the algorithm first constructs G_0, which contains only one node, i.e., the AC node. LevAdj then proceeds to construct the nodes in $\mathrm{G}_i (i \geqslant 1, i \in Z)$. From a high level, nodes in G_i choose among their neighbors in G_{i-1} and decide which one is the most appropriate. More specifically, given nodes a and b in the same level G_i, a is preferred over b when the parent of a has lighter reporting load than that of node b. In this way, LevAdj constructs the entire tree recursively including all the nodes.

From the initial construction of the tree, we get the reporting tree where nodes at each level strive to select parent nodes based on their parents reporting load. To further improve the performance of such a greedy selection, we employ the adjustment phase on the initially constructed tree. From a high level, such an adjustment involves moving a whole subtree from its parent node to another parent to get a better result (the busiest node reporting load is reduced). The recursive steps of adjusting the tree are as follows.

- The adjustment begins from the nodes in G_2.
- For each node in G_2, if the busiest node load is reduced by swapping its subtree with the subtree of some other node in the same level, the algortihm makes the swapping.
- If all the nodes in G_2 have been processed, then the algorithm proceeds to process $\mathrm{G}_3, \mathrm{G}_4, \ldots$ in turn until reaching the leaf nodes.
- If there is any change to the current topology, we restart the adjustment process from nodes in G_2.

In the following, we prove the above adjustment process can terminate.

Convergence Proof: We use $G_i[j]$ to represent the reporting load of the jth node in G_i. Without loss of generality, a node n in G_{i+1} chooses among his neighbors, say, nodes a and b, in G_i. Node a is preferred over b when the parent of node a has lighter reporting load than that of node b. Let $size(i)$ represent the number of nodes in G_i, and L refer to the number of levels. We use $a[0]$, $a[1]$, $\ldots$, $a[i]$ to stand for the path from node a to the root and $b[0]$, $b[1]$, $b[2]$ for node b. If node a is preferred over b, it means there exists an integer k such that the reporting load of $a[k]$ is smaller than that of $b[k]$ and the reporting load of $a[x]$ equates $b[x]$ for any $x \in [0, k-1]$.

When adjusting node n in G_{i+1}, if the algorithm moves its subtree from its original parent b to node a, it is easy to prove that new $D'(k)$ is smaller the original $D(k)$, where

$$D(i) = \sum_{k=0}^{size(i)} \left(size(i) \times G_i[k] - \sum_{k=0}^{size(i)} G_i[k] \right)^2$$

because the difference between node a's and node b's load will be smaller after the adjustment.

We can define a variable V to represent the reporting load in the topology as follows:

$$V = \{D(0), D(1), D(2), \ldots, D(L)\}.$$

We can compare such an array between its original value and its new value V', where

$$V' = \{D'(0), D'(1), D'(2), \ldots, D'(L)\}.$$

If we give more priority to $D(l_1)$ than $D(l_2)$ where $l_1 < l_2$, we can prove that after every adjustment, $V' < V$: $D(x) = D'(x)$ for any x in $[0, k-1]$ because the reporting load of $a[x]$ equates $b[x]$ for any x in $[0, k-1]$ and $D(k) < D'(k)$. From the definition, we know that $D(i)$ value is a nonnegative integer. It must decrease by some amount after one adjustment. Thus, there exists a limited number of adjustments, and hence the adjustment will always terminate at some stage.

As we show in Section VIII-C, LevAdj works well in reducing the busiest link load especially in DCNs according to our experiments.

B. BinDin Algorithm

While the LevAdj algorithm is tailored for DCNs with symmetric, layered topologies that are commonly seen in current practice, there still exist DCN topologies that do not conform to

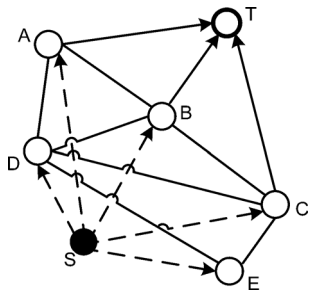


Fig. 6. Illustration of the BinDin algorithm. Node T refers to AC, node S refers to the dummy source, and other nodes are the reporting nodes (sources to send the traffic summaries). The maximum capacity of all the edges represented by solid lines is initialized to n and will be changed dynamically during the algorithm execution. The capacity of all the edges depicted as dotted lines is set to *1* and never changes during the algorithm execution.

such patterns. Hence, in this section we present a more general algorithm called BinDin.

Key Insights: To reduce the reporting load of the busiest link, some nodes should avoid using the busiest link to send traffic summaries. In the reporting phase, the AC is the sink, while other nodes are the sources. This circumstance is similar to the classic max-flow problem with only one sink, while it is different from the maximum-flow problem since our problem setting has multiple sources. To transform our problem to the max-flow problem, we add a dummy source that has a direct link with the other nodes. Adding the dummy source removes the difference between reporting traffic summaries to the AC and max-flow problem; but at the same time it raises a new issue: The dummy source will not go through all the servers to send information to the AC, i.e., not all the nodes will report traffic summaries to the AC, violating the requirements of the reporting phase in DFL. To ensure all the nodes are waypoints for the dummy source to send traffic summaries to the AC, we set the capacity of the direct link between dummy source and reporting nodes to *1*, which sums up to the quantities of influx flow to the sink. The capacity of the links in the original graph will be set to n, where n is the number of the reporting nodes. Fig. 6 depicts an example.

Algorithm Details: As shown in Fig. 6, S is the (dummy) source node, while T is the destination node (the AC). It forms a flow model based on an undirected graph. The maximum capacity of the edges corresponding to real links in the DCN is set to n, while the capacity of the edges between the dummy source S and other nodes is set to *1*. The key idea of BinDin is that it repeatedly makes use of the link capacity to diminish the reporting load of the busiest link. As shown in Fig. 7, we leverage the BinDin algorithm to obtain a viable flow distribution in which the value of the flows on all the edges between S and other nodes is 1. If such a flow distribution does exist, the algorithm remembers the load of the busiest link in this flow distribution, reduces the capacity of the edges by half, and then continues to minimize reporting load on the busiest link with capacity between *0* and $n/4$. Otherwise (if such a flow distribution does not exist), BinDin seeks to minimize the reporting load of the busiest link with the capacity between $n/2ph >$ to n. Further utilizing the methodology of binary search, we can run BinDin algorithm in O($log_2 n$) time. As the BinDin algorithm solves the maximum-flow problem in the general case with no more than Cn^2p operation, where n is the number of

```
while(max > min){
  mid = (max + min)/2;
    Dinic(mid);
    satisfy = Check();
  if(satisfy) {
    max = mid - 1;
  }
  else {
    min = mid + 1;
  }
}
```

Fig. 7. Pseudocode of the BinDin algorithm.

nodes of the DCN, p is the number of links in the DCN, and C is some constant not depending on the network [19]. Therefore, the overall time complexity of BinDin is O($n^2p\ \log_2(n)$).

In Section VIII-C, we compare the performance and overhead of LevAdj and BinDin, and find that the BinDin algorithm consumes less bandwidth than LevAdj while BinDin requires much more computation time.

VI. Detection

The AC performs consistency checks for each neighborhood $\mathbb{N}(r)$ based on the received traffic summaries. However, since an epoch may only have a small number of packets, detecting a suspicious neighborhood based on the consistency checks for *individual* epochs can introduce a large error rate. Consider an extreme example: If in an epoch, a neighborhood $\mathbb{N}(r)$ only transmits a single packet and the packet was spontaneously lost, concluding that the packet-loss rate is 100% and $\mathbb{N}(r)$ is suspicious is inaccurate. To deal with this problem, we still perform the consistency checks and estimate the discrepancy for individual epochs, but make the detection based on the *aggregated* discrepancies over a set of E epochs (called *accumulated epochs*), so that the total number of packets over the E epochs is more than a certain threshold N to give a high enough accuracy (e.g., $> 99.9\%$) on the detection results. Section VIII studies the value of N. Therefore, the AC stores the traffic summaries for each neighborhood and makes detection when the total number of packets N is reached. More specifically, let $n_x^{\leftarrow y}(e)$ and $n_x^{\rightarrow y}(e)$ denote the $n_x^{\leftarrow y}$ and $n_x^{\leftarrow y}$ in the traffic summary for epoch e, respectively; for a certain neighborhood $\mathbb{N}(r)$, whenever

$$\max\left\{\sum_e\sum_i n_i^{\rightarrow r}(e), \sum_e\sum_i n_i^{\leftarrow r}(e)\right\} > N \qquad (9)$$

(where $i \in \mathbb{N}(r)$ and e iterates over all the accumulated epochs), indicating N is reached, the AC performs the following checks to inspect if $\mathbb{N}(r)$ is suspicious.

Flow Conservation: The AC first extracts $n_i^{\rightarrow r}(e)$ and $n_i^{\leftarrow r}(e)$ for each node i in $\mathbb{N}(r)$ for each epoch e, and calculates the difference between the number of packets sent to r and the number of packets received from r over all the E accumulated epochs. If the ratio of the difference to the total number of packets in all the E accumulated epochs is larger than a threshold β, i.e.,

$$\frac{|\sum_e\sum_i n_i^{\rightarrow r}(e) - \sum_e\sum_i n_i^{\leftarrow r}(e)|}{\max\{\sum_e\sum_i n_i^{\rightarrow r}(e), \sum_e\sum_i n_i^{\leftarrow r}(e)\}} > \beta \qquad (10)$$

then the AC detects $\mathbb{N}(r)$ as suspicious. β is set based on the administrator's expectation of the natural packet-loss rate; e.g., in the simulations in Section VIII, we set β to be four times of the natural packet-loss rate in a neighborhood.

Content Conservation: The AC then extracts the sketches in the traffic summaries in $\mathbb{N}(r)$, and estimates the discrepancy δ_f between the sketches for packets sent to r and the sketches for packets received from r. The AC detects $\mathbb{N}(r)$ as malicious if δ_f is larger than a certain threshold, i.e.,

$$\delta_f > \frac{2\alpha\beta}{\alpha+\beta} \times \max\left\{\sum_e \sum_i n_i^{\rightarrow r}(e), \sum_e \sum_i n_i^{\leftarrow r}(e)\right\}$$

where

$$\delta_f = \| \cup_{i\in\mathbb{N}(r)} \mathcal{F}_{K_f^j}(\mathbb{C}_i^{\leftarrow r}) - \cup_{i\in\mathbb{N}(r)} \mathcal{F}_{K_f^j}(\mathbb{C}_i^{\rightarrow r})\|_2^2. \quad (11)$$

It has been proven [20] that the above threshold can satisfy the (α, β, δ)-accuracy defined in Section II-C.

Timing Consistency: The AC also extracts the difference between the average packet departure time and arrival time and concludes that $\mathbb{N}(r)$ is suspicious if the difference is larger than the expected upper bound on the 2-hop latency.

VII. Security Analysis

Due to the lack of space, We only show DFL's security against a single malicious node while DFL's security against colluding nodes can be similarly derived (and resembles the analysis in the previous work [41]).

Security Against Corrupting the Data Packets: Dropping, modifying, and fabricating data packets in a neighborhood $\mathbb{N}(m)$ will cause inconsistencies between sketches in $\mathbb{N}(m)$ as mentioned earlier. Delaying data packets in $\mathbb{N}(m)$ will cause abnormal deviation between average packet arrival and departure timestamps in $\mathbb{N}(m)$. If a malicious router changes the timestamps in data packets embedded by the source nodes, it is equivalent to modifying packets, and packets may be mapped to different epochs, in which case such an attack will manifest itself by causing inconsistencies in the sketches of a neighborhood containing the malicious router.

Security Against Corrupting $\mathsf{d_{ac}}$*:* As we mentioned earlier, if a malicious node m drops the $\mathsf{d_{AC}}$, some nodes adjacent to m will fail to send the correct traffic summaries to the AC, thus causing a neighborhood containing m to be detected. We note that the authentication of $\mathsf{d_{AC}}$ is needed. Otherwise, a malicious node can replace the sampling and fingerprinting keys with its own fake keys, by which the malicious node can predict the output of other nodes sketches and perform packet modification attacks. In addition, if the epoch IDs in $\mathsf{d_{AC}}$ were not authenticated, a malicious node can replace the oldest live epoch ID in $\mathsf{d_{AC}}$ for which the traffic summaries are requested with the current epoch ID. In this way, inconsistencies of traffic summaries can be detected for some *benign* neighborhood due to the packet transmission delay as Section IV-A describes. With the authentication of $\mathsf{d_{AC}}$, any attempt to modify $\mathsf{d_{AC}}$ will be detected (after $\lceil \frac{D}{L} \rceil$ epochs).

It is noteworthy that the $\mathsf{d_{AC}}$ sent at the end of epoch e cannot simply disclose the MAC secret key K_{e-1} for the previous epoch $e-1$. This is because at the time K_{e-1} is disclosed, the $\mathsf{d_{AC}}$ sent at the end of epoch $e-1$ may still have not reached certain nodes. Hence, a malicious node that has already received K_{e-1} might send K_{e-1} to a downstream colluding node via an out-of-band channel, so that the colluding node can break the authenticity of the $\mathsf{d_{AC}}$ sent in epoch $e-1$. Hence, at the end of an epoch e, we disclose the MAC key for epoch $e - \lceil \frac{D}{L} \rceil$ to ensure the $\mathsf{d_{AC}}$ sent in epoch $e - \lceil \frac{D}{L} \rceil$ has reached all the nodes in the network.

Security Against Corrupting the Reporting Messages: First, due to the use of the Onion Authentication, a malicious node m cannot *selectively* drop the reporting messages of a *remote* (non-adjacent) node r, to frame a neighborhood containing node r. Since all the accumulated reporting messages are "combined" at each hop, m can only drop the reporting messages from its *immediate* neighbors, which will manifest a neighborhood containing m as suspicious.

VIII. Performance Evaluation

We analyze the protocol overhead and study the detection efficiency of DFL via measurements and simulations with our implementation of the classic Sketch [5] in C++. In particular, we compare the tradeoffs between LevAdj and BinDin for the reporting phase.

A. Storage Overhead

DFL incurs only per-neighbor state while existing secure path-based FL protocols require per-source and per-path state. In this section, we quantify the per-neighbor storage overhead of a DFL router r, which primarily includes the packet cache and the sketch for each neighbor s.

Sketch Size: The sketch size based on (7) and (8) is less than 500 B.

Cache Size and Per-neighbor Storage Overhead: We now study the cache size for temporarily storing packet hashes in live epochs, which, together with the sketch size analyzed above, constitutes the per-neighbor storage overhead of a DFL router. We denote the upper bound of one-way network latency as D, epoch length as L, and the number of packets per second as η. Using 20-B packet hashes, the cache size is given by

$$\left\lceil \frac{D}{L} + 1 \right\rceil \times 20 \cdot \eta \cdot L. \quad (12)$$

We omit the 1-bit indicator for each packet hash entry to indicate to which packet stream the packet belongs (see Fig. 3). Assuming the per-neighbor sketch size is 500 B, one-way latency $D = 20$ ms, and the average packet size is 300 B for an OC-192 link, we derive the per-neighbor storage overhead of a DFL router with different epoch lengths shown in Fig. 8. We can observe that, with an epoch length of 20 ms, only around 4 MB is required per neighbor. The "humps" exist in the curve due to the use of the ceiling function in (12).

B. Key Management Overhead

One distinct advantage DFL presents is that each router in DFL shares only one secret key with the AC, whereas in path-based FL protocols it is *necessary* for each router to share a secret key with each source node in the network in the worst case [12], which dramatically complicates the key management and broadens the vulnerability surface. To quantify DFL advantage over path-based FL protocols, we leverage the data center topologies from DCell [24], BCube [23], and VL2 [22], and measured ISP topologies from the Rocketfuel dataset [37] and the topology from Internet2 [3]. Fig. 9 shows the maximum

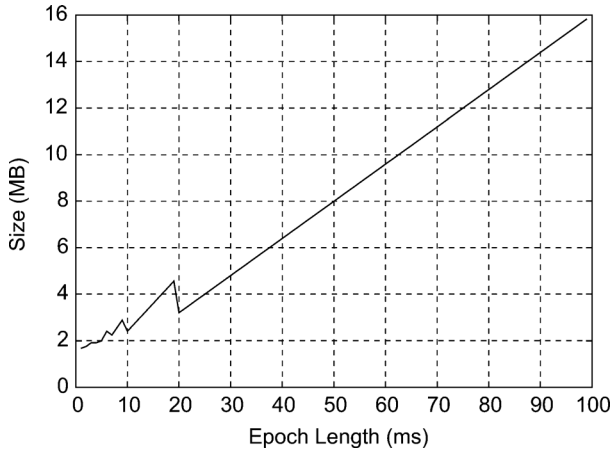


Fig. 8. Router per neighbor for an OC-192 link with the average packet size of 300 B and one-way network latency as 20 ms.

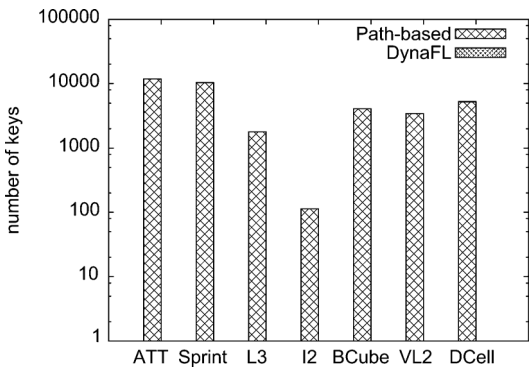


Fig. 9. Key management overhead at each router. A router in DFL always requires just one key shared with the AC .

number of keys each router needs to manage in path-based FL protocols; and a router in DFL always requires only one secret key shared with the AC (thus invisible in the figure). We can see that the number of keys a router needs to manage in path-based FL is 100–10 000 times higher than that in DFL.

C. Reporting Overhead

Reporting traffic summary consumes bandwidth, and it takes time to generate reporting paths (which matters when topology changes frequently). We evaluate the bandwidth overhead and computational overhead of DFL—more specifically, LevAdj and BinDin.

Bandwidth Overhead: We analyze the bandwidth consumption on each link through the reporting traffic summaries based on recently proposed popular datacenter topologies from BCube [23], DCell [24], Fat-tree [33], and VL2 [22]. We study both LevAdj and BinDin and analyze their tradeoffs.

For the datacenter topologies with 3636–6144 switches/routers and 3456–5256 servers, we select one of the top-tier switches as the AC. We consider the epoch length $L = 20$ ms, a per-neighbor traffic summary of 500 B, and the epoch sampling rate to 0.1%. The bandwidth consumptions are shown in Figs. 10–12. Assuming OC-192 links, we can see that the bandwidth used for reporting traffic summaries on a link is under 0.04% of a link capacity. These figures also show both LevAdj and BinDin reduce the bandwidth consumption by around

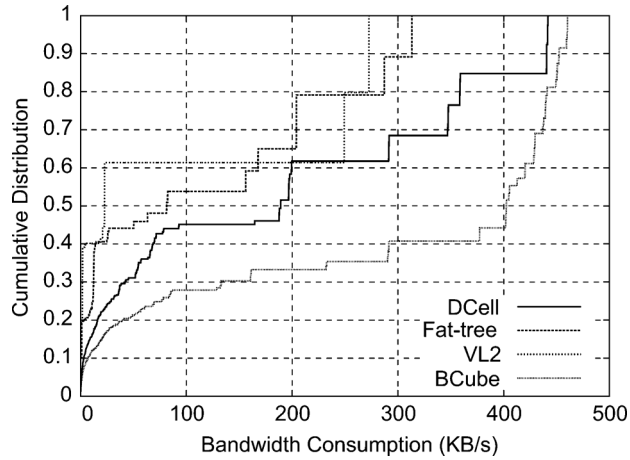


Fig. 10. Datacenter topologies with epoch sampling rate = 0.1% based on minimum spanning tree.

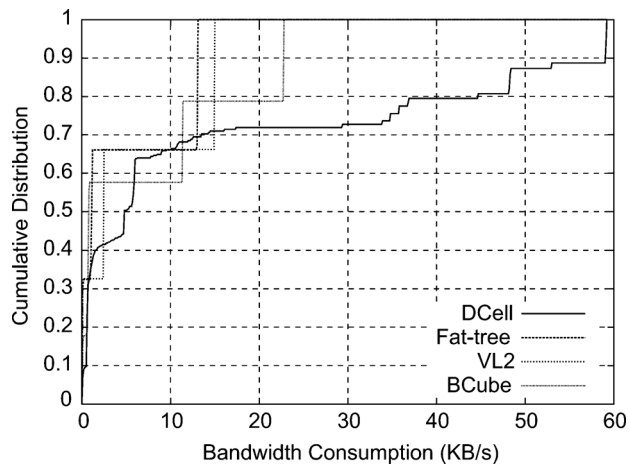


Fig. 11. Datacenter topologies with epoch sampling rate = 0.1% based on LevAdj algorithm.

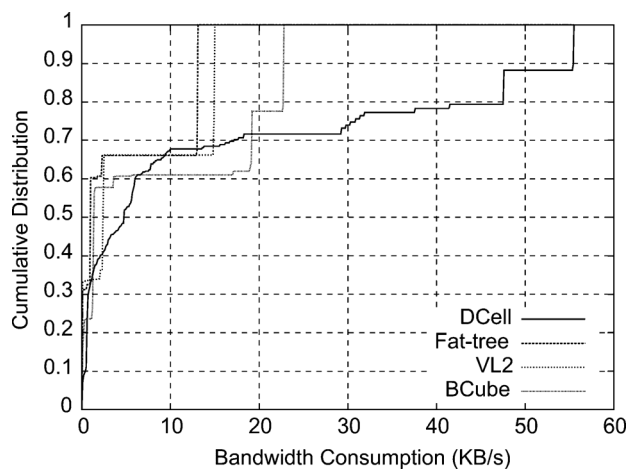


Fig. 12. Datacenter topologies with epoch sampling rate = 0.1% based on BinDin algorithm.

10 times in DCN topologies, and LevAdj algorithm performs almost as well as BinDin algorithm. From the results of the bandwidth overhead, we can see that the difference between the two algorithms is inconsiderable. More specifically, the results

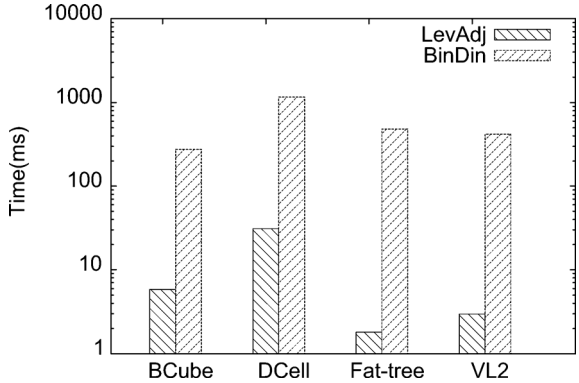


Fig. 13. Comparison of DCNs time overhead consumption between LevAdj algorithm and BinDin algorithm.

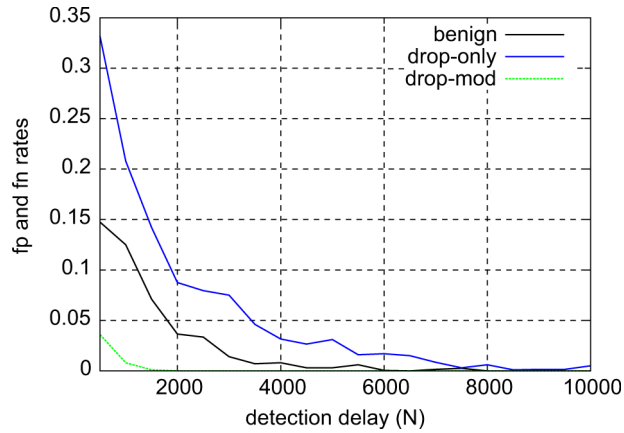


Fig. 14. False positive and negative rates.

differ little for DCN topologies. This is because the LevAdj algorithm assumes that all nodes choose the shortest paths to the root (the AC), which works well when the topology is symmetric and layered, where nodes have multiple equal-length paths between each other. All the datacenter topologies used in this simulation conform to such patterns.

Computational Overhead: From the above results, we observe that the bandwidth consumption between BinDin algorithm and LevAdj algorithm is insignificant. Thus, we further analyze the computational overhead of the two algorithms both in DCN topologies with 3636–6144 switches/routers and 3456–5256 servers. Our experiment environment is based on Intel Core Duo CPU E7500 2.93 GHz with 4 GB RAM. Fig. 13 plots the computation time of the two algorithms. From these results, we can clearly see that the running time of LevAdj is about a hundred times faster than that of BinDin.

D. Detection Delay

As Section VI states, the AC performs consistency checks and detects any anomalies only when the total number of packets over multiple epochs is accumulated more than a certain threshold N in order to give a high enough accuracy (e.g., $> 99.9\%$) on the detection results. Hence, the number of packets N characterizes the detection delay of the FL protocol. We fully implement the classic Sketch due to Alon *et al.* [5] in C++ with a fourwise independent hash function and perform simulations to study N.

Since in DFL, neighborhoods are inspected by the AC *independently*, we also perform simulations for independent neighborhoods with different sizes. Since we showed DFL's security against colluding attacks in Section VII, we emulate a single malicious node in our simulations. Our setting is as follows. The natural packet-loss rate in a neighborhood is 0.001, and the detection thresholds for both flow conservation and content conservation are $\beta = 2\alpha = 0.004$. "benign" in Fig. 14 depicts the false positive rates in benign cases where no malicious routers exist in the neighborhood. We can see that with $N > 5000$ packets, the false positive rate is under 1%. "drop-only" shows the false negative rates with a malicious router that only drops packets with a probability of 0.005. "drop-mod" plots the false negative rates with a malicious router that both drops and modifies packets with a probability of 0.005, respectively. We can see that the sketch-based approach is effective in detecting packet modification attacks since by modifying packets the malicious router is detected faster in "drop-mod" than in "drop-only."

Compared to the previous work DynaFL, our new reporting algorithms (LevAdj and BinDin) not only improve FL scalability by reducing the bandwidth consumption, but also reduce the detection delay. For example, in DynaFL, to save bandwidth consumption, it only samples a fraction of epochs for sending the traffic summaries. This fraction may be tiny (e.g., the sampling rate is 0.1% in DCN) in order to keep the bandwidth consumption low. In DFL, since our reporting algorithms lessen the bandwidth consumption by an order of magnitude compared to DFL, we can increase the sampling rate by about 10 times, without worrying too much about the overhead. In this way, DFL localizes faults (if any) in one tenth of time used in DynaFL.

IX. Related Work

Recent research on datacenters focuses on improving the network scalability and performance, while the security of the network infrastructure has not been well studied [17]. There exist multiple *secure* FL proposals [9], [12], [40], [42], which are all path-based. They are inapplicable to enterprise and datacenter networks since they fail to support dynamic routing paths, require per-path state at routers, and incur per-source key sharing and management. Besides these fundamental limitations, we show that most existing FL protocols also suffer from security vulnerabilities.

For example, WATCHERS [14], [25], AudIt [7], and Fatih [32] implement the traffic summaries using either counters or Bloom Filters [13] with no secret keys, thus remaining vulnerable to packet modification attacks as Section III-D shows.

Both ODSBR [10], [11] and Secure Traceroute [35] activate FL only when the end-to-end packet-loss rate exceeds a certain threshold. However, a malicious node can safely drop packets when FL is not activated and behave "normally" when FL is invoked. In addition, ODSBR does not consider natural packet loss, which can make the algorithm either not converge or incur high false positives by incriminating benign links.

Liu *et al.* propose enabling 2-hop-away routers in the path to monitor each other [28] by using 2-hop acknowledgment packets. However, such a 2-hop-based detection scheme is vulnerable to colluding neighboring routers. Similarly, both

Watchdog [30] and Catch [29] can identify and isolate malicious routers for wireless ad hoc networks, where a sender S verifies if the next-hop node f_i indeed forwards S's packets by *promiscuously* listening to f_i's transmission. Both Watchdog and Catch are vulnerable to collusion attacks, where a malicious node f_m drops the packets of a remote sender S (which is out of the promiscuous listening range of f_m) while the colluding neighbors in the promiscuous listening range of f_m intentionally do not report the packet dropping behavior of f_m.

In this paper, DFL builds on our previous work (DynaFL), which first proposed the neighborhood-based FL approach and utilized the delayed function disclosure mechanism [41]. However, DynaFL targets general network topologies and fails to present an efficient reporting algorithm. As a result, DynaFL suffers high bandwidth consumption and long detection delay. By leveraging the unique opportunities given by DCN topologies, DFL makes neighborhood-based FL truly practical by significantly reducing both the bandwidth consumption and detection delay.

X. Conclusion

We design practical and scalable FL protocols for enterprise and datacenter networks that can cope with dynamic traffic patterns with constant, small router state. After identifying the fundamental limitations of previous FL protocols, we explore a neighborhood-based FL approach and propose DFL, which is applicable to general network topologies. While existing path-based FL protocols aim to identify a specific faulty *link*, DFL localizes faults to a coarser-grained 1-hop neighborhood to achieve four distinct advantages. First, DFL does not require any minimum duration time of paths or flows as path-based FL protocols do. Hence, DFL can cope with short-lived flows popularly seen in datacenter networks. Second, in DFL, a source node does not need to know the exact outgoing path in contrast to path-based FL. Hence, DFL can support agile (e.g., packet-level) load balancing such as in VL2 [22]. Third, a DFL router only needs around 4 MB per neighbor state, while a router in a path-based FL protocol requires per path state. Finally, a DFL router only maintains a single secret key shared with the AC, while a path-based FL protocol requires per-source key storage at routers. On the other hand, the neighborhood-based protocols require a central administrator and cannot detect if a malicious router targets a single node's traffic. Consequently, it remains an open research challenge to design a FL protocol that operates in intradomain settings across mutually distrusting administrative boundaries.

References

[1] K. Zetter, "Cisco security hole a whopper," *Wired* 2005 [Online]. Available: http://www.wired.com/politics/security/news/2005/07/68328

[2] T. Espiner, "Symantec warns of router compromise," 2008 [Online]. Available: http://news.cnet.com/Symantec-warns-of-router-compromise/2100-7349_3-6227502.html

[3] "The Internet2 observatory: Proposal process," 2008 [Online]. Available: http://www.internet2.edu/observatory/archive/proposal-process.html

[4] D. Achlioptas, "Database-friendly random projections," in *Proc. 20th ACM SIGMOD-SIGACT-SIGART PODS*, New York, NY, USA, 2001, pp. 274–281.

[5] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proc. STOC*, 1996, pp. 20–29.

[6] X. Ao, "Report on DIMACS workshop on large-scale Internet attacks," 2003 [Online]. Available: http://dimacs.rutgers.edu/Workshops/Attacks/internet-attack-9–03.pdf

[7] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker, "Loss and delay accountability for the Internet," in *Proc. IEEE ICNP*, 2007, pp. 194–205.

[8] K. Argyraki, P. Maniatis, and A. Singla, "Verifiable network-performance measurements," in *Proc. ACM CoNEXT*, 2010, Art. no. 1.

[9] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy, "Highly secure and efficient routing," in *Proc. IEEE INFOCOM*, 2004, pp. 197–208.

[10] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens, "ODSBR: An on-demand secure Byzantine resilient routing protocol for wireless ad hoc networks," *Trans. Inf. Syst. Security*, vol. 10, no. 4, 2008, Art. no. 6.

[11] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, "An on-demand secure routing protocol resilient to Byzantine failures," in *Proc. ACM WiSE*, 2002, pp. 21–30.

[12] B. Barak, S. Goldberg, and D. Xiao, "Protocols and lower bounds for failure localization in the Internet," in *Proc. EUROCRYPT*, 2008, pp. 341–360.

[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[14] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson, "Detecting disruptive routers: A distributed network monitoring approach," in *Proc. IEEE Symp. Security Privacy*, May 1998, pp. 115–124.

[15] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A protection architecture for enterprise networks," in *Proc. USENIX Security*, 2006, Art. no 10.

[16] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoret. Comput. Sci.*, vol. 312, no. 1, pp. 3–15, 2004.

[17] K. Chen, C. Hu, X. Zhang, K. Zheng, Y. Chen, and T. Vasilakos, "Survey on routing in data centers: Insights and future directions," *IEEE Netw.*, vol. 25, no. 4, pp. 6–10, Jul.–Aug. 2011.

[18] I. Cunha, R. Teixeira, and C. Diot, "Measuring and characterizing end-to-end route dynamics in the presence of load balancing," in *Proc. PAM*, 2011, pp. 235–244.

[19] E. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math. Dokl*, vol. 11, pp. 1277–1280, 1970.

[20] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford, "Path-quality monitoring in the presence of adversaries," in *Proc. SIGMETRICS*, 2008, pp. 193–204.

[21] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, 2005.

[22] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.

[23] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM*, 2009, pp. 63–74.

[24] C. Guo, H. Wu, K. Tan, L. Shiy, Y. Zhang, and S. Luz, "Dcell: A scalable and fault-tolerant network structure for data centers," in *Proc. ACM SIGCOMM*, 2008, pp. 75–86.

[25] J. R. Hughes, T. Aura, and M. Bishop, "Using conservation of flow as a security mechanism in network protocols," in *Proc. IEEE Symp. Security Privacy*, 2000, pp. 132–141.

[26] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed diagnosis in enterprise networks," in *Proc. ACM SIGCOMM*, 2009, pp. 243–254.

[27] C. Labovitz, A. Ahuja, and M. Bailey, "Shining light on dark address space," Arbor Networks, Waltham, MA, USA, Tech. rep., 2001.

[28] K. Liu, J. Deng, P. K. Varshney, and K. Balakrishnan, "An acknowledgment-based approach for the detection of routing misbehavior in MANETs," *IEEE Trans. Mobile Comput.*, vol. 6, no. 5, pp. 536–550, May 2007.

[29] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan, "Sustaining cooperation in multi-hop wireless networks," in *Proc. USENIX NSDI*, 2005, pp. 231–244.

[30] S. Marti, T. J. Giuli, K. Lai, and M. Baker, "Mitigating routing misbehavior in mobile ad hoc networks," in *Proc. ACM MobiCom*, 2000, pp. 255–265.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
[32] A. T. Mizrak, Y. c. Cheng, K. Marzullo, and S. Savage, "Fatih: Detecting and isolating malicious routers," *IEEE Trans. Depend. Secure Comput.*, vol. 3, no. 3, pp. 230–244, Jul.–Sep. 2005.
[33] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, and V. Subram, "PortLand: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. ACM SIGCOMM*, 2009, pp. 39–50.
[34] Y. Otachi, H. L. Bodlaender, and E. J. v. Leeuwen, "Complexity results for the spanning tree congestion problem," in *Proc. Workshop Graph Theoret. Concept Comput. Sci.*, 2010, pp. 3–14.
[35] V. N. Padmanabhan and D. R. Simon, "Secure traceroute to detect faulty or malicious routing," *Comput. Commun. Rev.*, vol. 33, no. 1, pp. 77–82, 2003.
[36] A. Perrig, R. Canetti, D. Song, and D. Tygar, "The TESLA broadcast authentication protocol," *Cryptobytes*, vol. 5, no. 2, pp. 2–13, 2002.
[37] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, 2002, pp. 133–145.
[38] R. Thomas, "ISP security bof, nanog 28," 2003 [Online]. Available: http://www.nanog.org/meetings/nanog28/presentations/thomas.pdf
[39] M. Thorup and Y. Zhang, "Tabulation based 4-universal hashing with applications to second moment estimation," in *Proc. SODA*, 2004, pp. 615–624.
[40] X. Zhang, A. Jain, and A. Perrig, "Packet-dropping adversary identification for data plane security," in *Proc. ACM CoNEXT*, 2008, Art. no 24.
[41] X. Zhang, C. Lan, and A. Perrig, "Secure and scalable network fault localization under dynamic traffic patterns," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 317–331.
[42] X. Zhang, Z. Zhou, G. Hasker, A. Perrig, and V. Gligor, "Network fault localization with small TCB," in *Proc. IEEE ICNP*, 2011, pp. 143–154.
[43] X. Zhang, Z. Zhou, H.-C. Hsiao, A. Perrig, and P. Tague, "Shortmac: Efficient data plane fault localization," in *Proc. NDSS*, 2012.

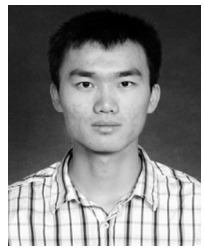
Xin Zhang received the B.S. degree in automation from Tsinghua University, Beijing, China, in 2006, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 2012.

He is currently is a Software Engineer with Google, Pittsburgh, PA, USA, working on datacenter cluster management. His research interests revolve around security and network.

Dr. Zhang received the Chinese Government Award for Outstanding Students Abroad in 2011.

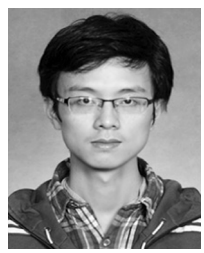
Fanfu Zhou received the B.S. degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2008, and the M.S. degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2012, and is currently pursuing the Ph.D. degree in computer science at Shanghai Jiao Tong University.

His research interests are primarily in computer network security and mobile network security.



Xinyu Zhu received the B.S. degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012, and is currently a postgraduate student in software engineering at Shanghai Jiao Tong University.

His interests lie broadly in network.



Haiyang Sun received the B.S. degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012, and is currently a postgraduate student with the School of Software, Shanghai Jiao Tong University.

His main interests include binary translation and virtualization.

Adrian Perrig (M'00–SM'12) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 2002.

He is a Distinguished Fellow with CyLab and a Full Professor of computer science with the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. He is formerly a Professor in electrical and computer engineering, engineering and public policy, and computer science with Carnegie Mellon University. His research revolves around building secure systems and includes network security, trustworthy computing, and security for social networks.

Athanasios V. Vasilakos (M'00–SM'11) received the Ph.D. degree in computer engineering from the University of Patras, Patras, Greece, in 1988.

He is currently a Professor with the University of Western Macedonia, Kozani, Greece. He has authored or coauthored over 200 technical papers in major international journals and conferences. He is the author/coauthor of five books and 20 book chapters in the areas of communications.

Prof. Vasilakos has served as General Chair, Technical Program Committee Chair, or TPC member (i.e., INFOCOM, SECON, MobiHoc) for many international conferences. He served or is serving as an Editor or/and Guest Editor for many technical journals, such as the IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT; IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART B: CYBERNETICS; IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE; IEEE TRANSACTIONS ON COMPUTERS, *ACM Transactions on Autonomous and Adaptive Systems*; the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS special issues of May 2009, January 2011, and March 2011; the *IEEE Communications Magazine*; *Wireless Networks*; and *Mobile Networks and Applications*. He is the founding Editor-in-Chief of the *International Journal of Autonomous and Adaptive Communications Systems* and *International Journal of Arts and Technology*. He is also General Chair of the Council of Computing of the European Alliances for Innovation.

Haibing Guan (M'00) received the Ph.D. degree in artificial intelligence from Tongji University, Shanghai, China, in 1999.

He is currently a Professor with the School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University, Shanghai, China, and the Director of the Shanghai Key Laboratory of Scalable Computing and Systems. His research interests include distributed computing, network security, network storage, green IT, and cloud computing.